Design, Implementation, and Evaluation of Task Management in Distributed Fault-Tolerant Real-Time Systems

Project Investigator: *Chao-Ju (Jennifer) Hou*

Graduate Students: *Bin Wang, Hung-ying Tyan, and Yi Ye*

Dept. of Electrical Engineering

The Ohio State University

Columbus, OH 43210-1272

jhou@ee.eng.ohio-state.edu

http://eewww.eng.ohio-state.edu/drcl

Outline of Presentation

- *Project Overview*
  - Real-time task system
  - Task management in real-time task systems
  - Software implementation
- *Fault-tolerance Components in Our Project*
  - Replication of critical modules
  - Primary and backup workstations for task transfer
  - Checkpointing and rollback recovery

Real-Time Task System

Every task is characterized by a laxity -- the latest time a task must start execution in order to meet its deadline.

- *Periodic tasks*
  - Invoked at fixed time intervals.
  - Attributes are usually known *a priori*.
- *Aperiodic tasks*
  - Invoked randomly in response to environmental stimuli.
  - Attributes are not completely specified.

Management of Real-Time Task Systems

- The execution of both periodic and aperiodic tasks must be
  - logically correct.
  - completed before their deadlines.
- Performance is assessed on a *per-task* basis.
- The *probability of dynamic failure* defined as the probability of a task failing to be completed in time, is used as performance metric.

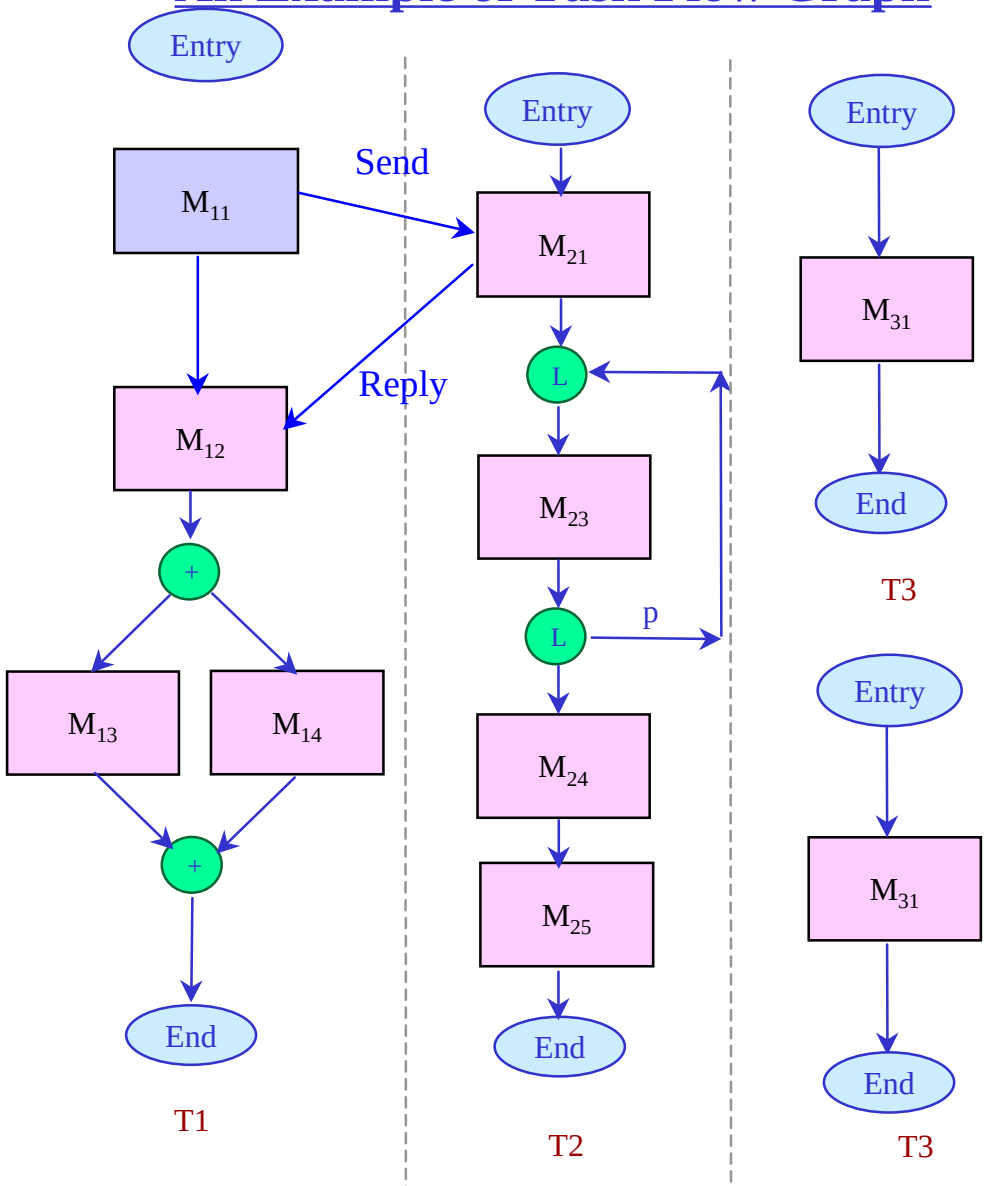
Project Objective

- We design, implement, and empirically evaluate a task management system in distributed real-time environments to meet the *timeliness* and *logical correctness* requirements of both periodic and aperiodic tasks.
- The project is a combination of two synergistic components: *scheme development* in a well-defined analytic framework and *validation with software system building* and experiments.

Methodology Used

- Task decomposition: Decompose periodic tasks into a set of communicating modules, and represent them by a task flow graph.
- Module allocation: Allocate periodic task modules to workstations subject to precedence constraints and timing requirements.
- Load redistribution: Dynamically redistribute aperiodic tasks as they arrive to minimize the probability of dynamic failure.
- Scheduling: Schedule modules/tasks on a node using the rate-monotonic policy, the earliest-deadline-first policy, or variations thereof.

An Example of Task Flow Graph



THE OHIO STATE UNIVERSITY

Tasks Performed

- Design task allocation and load redistribution schemes.
- Incorporate fault tolerance capabilities by
  - identifying and replicating *critical* modules.
  - taking advantage of *checkpointing and rollback recovery* techniques.
  - coordinating workstations to restart checkpointed processes in case of failure.
- Currently implement the proposed schemes as a software layer that lies between OS and application programs to empirically measure the performance.

Technical Approaches

- We devise a module allocation scheme to allocate periodic task modules in a *planning cycle* so that
  - the probability of completing each task with *both* logical and timing correctness is maximized,
  - task precedence and timing constraints are satisfied.
- We characterize load sharing with three component policies: the *transfer policy*, the *location policy*, and the *information policy*, and reduce the possibilities of

  (1) transferring an overflow task to an "incapable node,"

  (2) multiple nodes sending their overflow tasks to the same node;

  (3) excessive task transfers;

  (4) excessive communication and time overheads.

Module Replication for Fault Tolerance

Given a task flow graph that describes the computation and communication modules and the precedence and timing constraints among them, we consider

- which modules are replicated;
- how many copies are replicated for each selected module;
- how to assign the replicas to workstations;
- how to schedule the replicas on each workstation.

with the objective of achieving timely correctness.

Critical Path Analysis

- Observation: There is no need to replicate modules that are subject to less stringent timing requirements.
- Criterion for selecting critical modules:

$$LC_i - r_i < e_i + t_r ,$$

then M_i may not be completed in time in the case of failure.

where

- LC_i is the latest completion time of module M_i,
- r_i is the earliest release time of M_i,
- e_i is the execution time of M_i,
- t_r is the worst-case error recovery time.

Critical Path Analysis

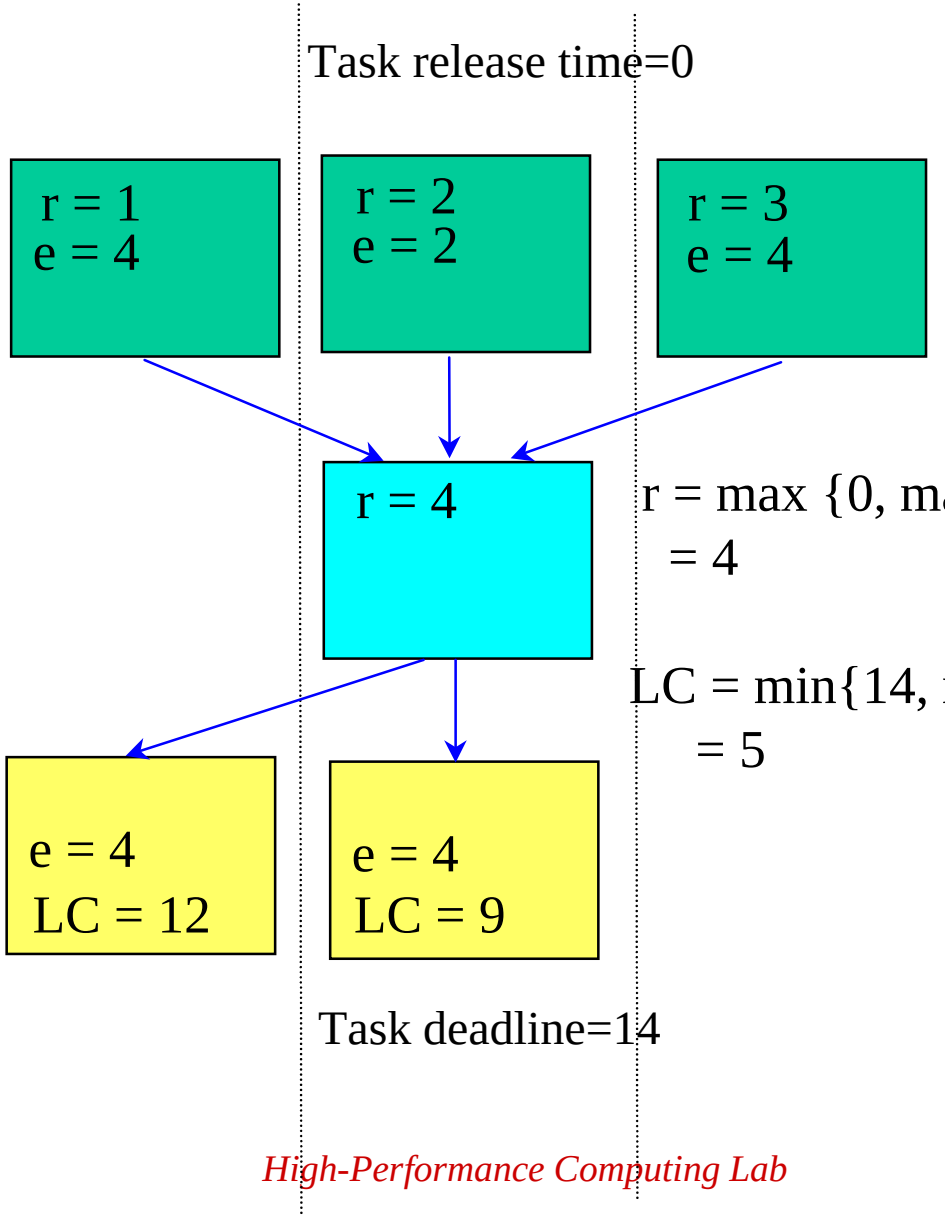
- Key Step 1: Calculate r_i from (1) the invocation time of the task and (2) the precedence constraints preceding M_i.
- Set r_i initially to the invocation time of the task to which M_i belongs. Then, modify $r_i$as

$$r_i = \max \{ r_i , \max_j \{ r_j + e_j: M_j \text{ --> } M_i \} \}$$

- Key Step 2: Calculate LCi from (1) the deadline of the task and the precedence constraints after Mi.
- Initially set LC_i to the deadline of the task to which M_i belongs. Then, modify LC_i as

$$LC_i = \min \{ LC_i , \max_j \{ LC_j - e_j: M_i \text{ --> } M_j \} \}$$

Example of Critical Path Analysis



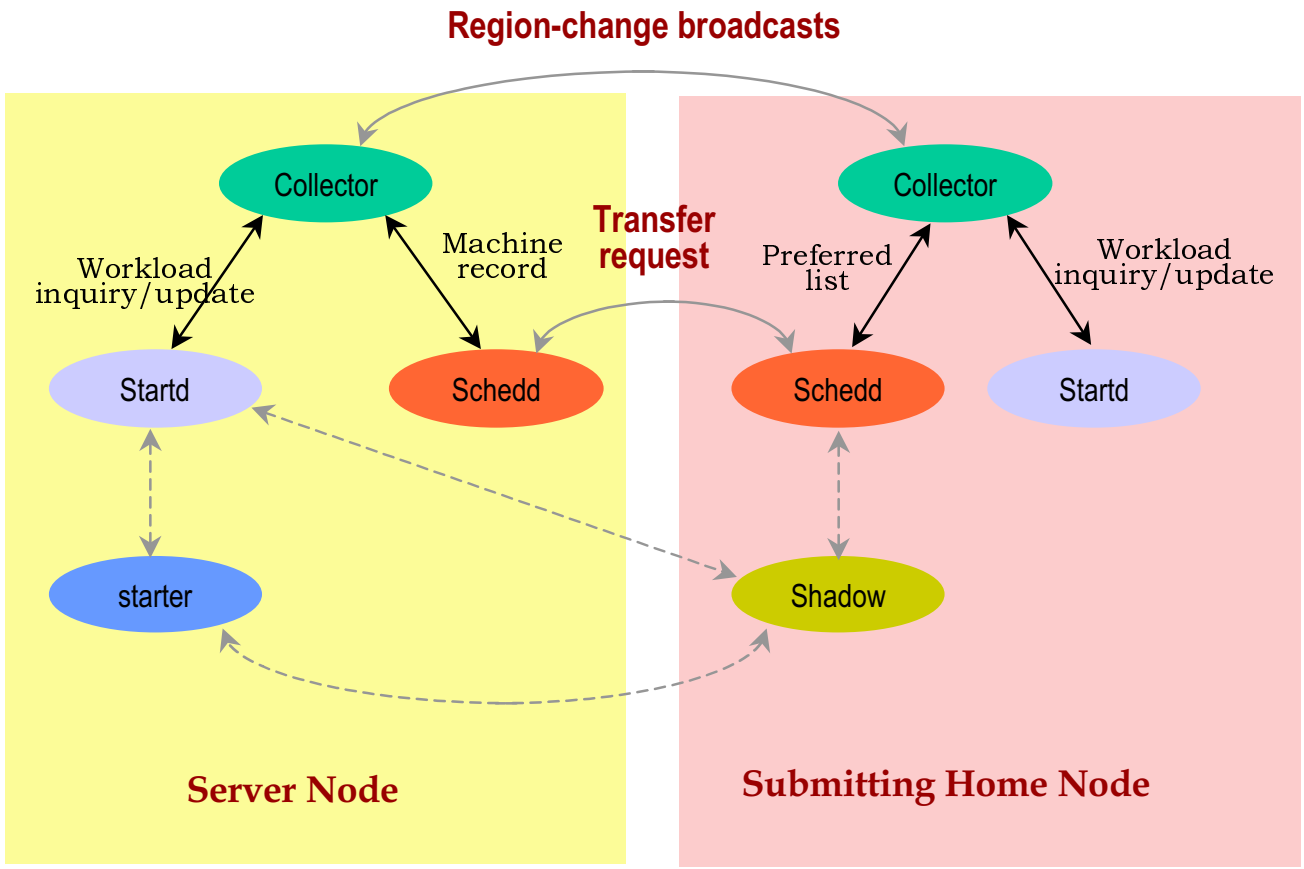
Determination of #Replicas

- There is a tradeoff between fault tolerance and timing requirements:
  - The larger #replicas, the better fault-tolerance capability.
  - Excessive replicas may jeopardize the timely completion of modules.
- We augment the task system with m replicas for each selected critical module, and use the module allocation scheme, coupled with the module scheduling algorithm, to determine the assignment and scheduling of all modules.
- If there is computation power left, try to increase #replicas until the required probability of dynamic failure is violated.

Software Configuration

- We implement the first version as a software layer outside the OS kernel at the user level, since this design
  - eliminates the need to access/change the internals of OS,
  - allows us to concentrate on varying the degree of design complexity and
  - is portable and can be ported to any POSIX-compliant platforms.
- We configure the proposed mechanism into three daemons, *Collector*, *Schedd*, *Startdd*. Two additional processes, *Shadow* and *Starter*, run on the submitting node and the server node, respectively, when a task is remotely executed.

Daemon Configuration



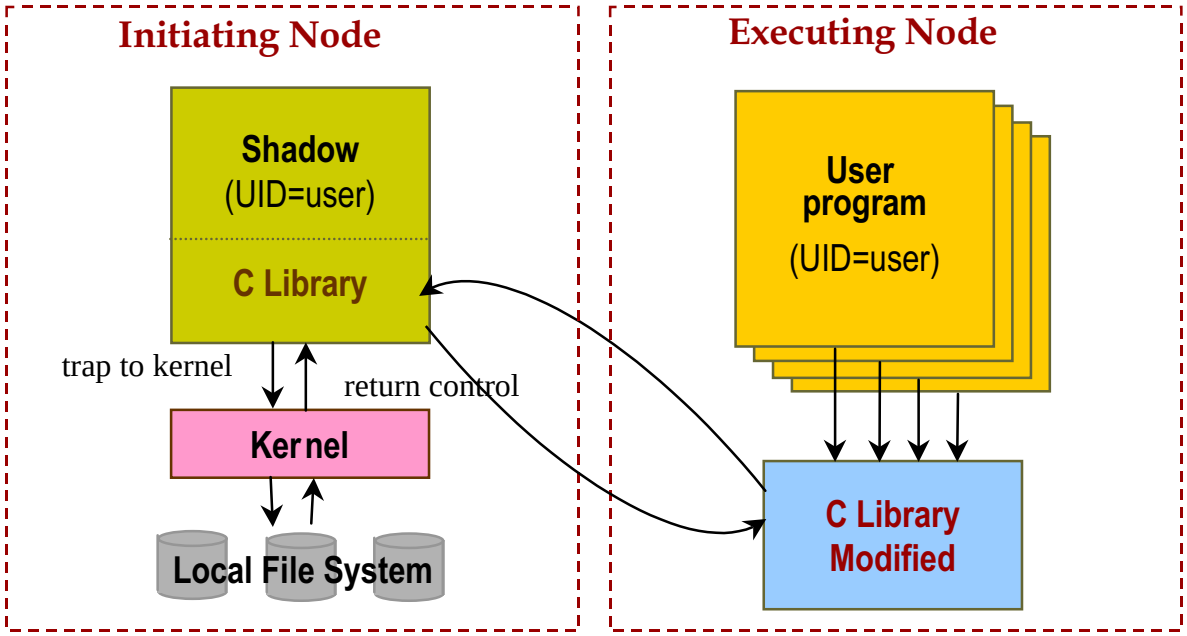
Fault-Tolerance and Security Features

- Both module allocation and load sharing are performed *transparently* to users.
- No code change is needed for user programs; only a relink to the modified C library is required for user programs.
- We preserve local execution environment for remotely executing processes via *remote system call mechanism*.
- We set protection for local file systems; they will not be touched by remotely executing tasks.
- We design a checkpointing scheme that *dynamically* varies checkpoint interval with respect to message passing frequency to reduce process rollback propagation.
- Processes are checkpointed at the end of each checkpoint interval and restarted at *backup* workstations whenever needed.

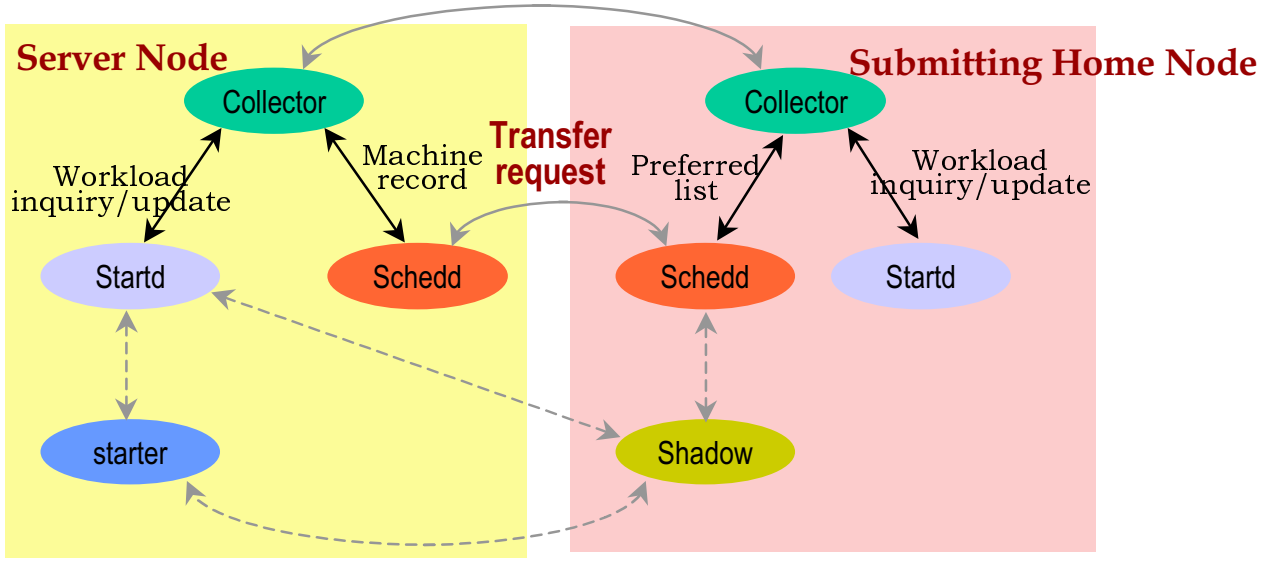
Remote System Calls

All environment-related system calls issued by a remote executing task are

- trapped by the *modified* C system call stubs, and
- forwarded to the *Shadow* on the home node which acts as an agent and executes the system calls on behalf of the task.



THE OHIO STATE UNIVERSITY



- The state of a process is transferred in the form of checkpointing files.
- Starter causes a running task to checkpoint by sending it the signal SIGTSTP.
- Starter sends the checkpoint file to Shadow which will restart the checkpointing file at a backup workstation in case of server workstation failure.
- We design a *location policy* which
  - avoids the situation of multiple nodes sending their overflow tasks to the same node;
  - selects, with the criteria of timely correctness and load balance, a backup workstation for executing the checkpointing file.